Constructing Call Graphs of Scala Programs

Karim Ali[1], Marianna Rapoport[1], Ondřej Lhoták[1], Julian Dolby[2], and Frank Tip[1]

[1] University of Waterloo
{karim, mrapoport, olhotak, ftip}@uwaterloo.ca
[2] IBM T.J. Watson Research Center
dolby@us.ibm.com

Abstract. As Scala gains popularity, there is growing interest in programming tools for it. Such tools often require call graphs. However, call graph construction algorithms in the literature do not handle Scala features, such as traits and abstract type members. Applying existing call graph construction algorithms to the JVM bytecodes generated by the Scala compiler produces very imprecise results due to type information being lost during compilation. We adapt existing call graph construction algorithms, Name-Based Resolution (RA) and Rapid Type Analysis (RTA), for Scala, and present a formalization based on Featherweight Scala. We evaluate our algorithms on a collection of Scala programs. Our results show that careful handling of complex Scala constructs greatly helps precision and that our most precise analysis generates call graphs with 1.1-3.7 times fewer nodes and 1.5-18.7 times fewer edges than a bytecode-based RTA analysis.

1 Introduction

As Scala [20] gains popularity, the need grows for program analysis tools for it that automate tasks such as refactoring, bug-finding, verification, security analysis, and whole-program optimization. Such tools typically need call graphs to approximate the behavior of method calls. Call graph construction has been studied extensively [11,21]; algorithms vary primarily in how they handle indirect function calls. Several Scala features such as traits, abstract type members, and closures affect method call behavior. However, to our knowledge, no call graph construction algorithms for Scala have yet been proposed or evaluated.

One could construct call graphs of Scala programs by compiling them to JVM bytecode, and then using existing bytecode-based program analysis frameworks such as WALA [15] or SOOT [29] on those generated bytecodes. However, as we shall demonstrate, this approach is not viable because significant type information is lost during the compilation of Scala programs, causing the resulting call graphs to become extremely imprecise. Furthermore, the Scala compiler translates certain language features using hard-to-analyze reflection. While solutions exist for analyzing programs that use reflection, such approaches tend to be computationally expensive or they make very conservative assumptions that result in a loss of precision.

Therefore, we explore how to adapt existing call graph construction algorithms for Scala, and we evaluate the effectiveness of such algorithms in practice. Our focus is on adapting low-cost algorithms to Scala, in particular Name-Based Resolution (RA) [26], Class Hierarchy Analysis (CHA) [9], and Rapid Type Analysis (RTA) [6]. We consider how key Scala features such as traits, abstract type members, and closures can be accommodated, and present a family of successively more precise algorithms. In a separate technical report [4], we formally define our most precise algorithm for FS_{alg}, the “Featherweight Scala” subset of Scala that was previously defined by Cremet et al. [8], and prove its correctness by demonstrating that for each execution of a method call in the operational semantics, a corresponding edge exists in the constructed call graph.

Our new algorithms differ primarily in how they handle the two key challenges of analyzing Scala: *traits*, which encapsulate a group of method and field definitions so that they can be mixed into classes, and *abstract type members*, which provide a flexible mechanism for declaring abstract types that are bound during trait composition. We implement our algorithms in the Scala compiler, and compare the number of nodes and edges in the call graphs computed for a collection of publicly available Scala programs. In addition, we evaluate the effectiveness of applying the RTA algorithm to the JVM bytecodes generated by the Scala compiler. For each comparison, we investigate which Scala programming idioms result in differences in cost and precision of the algorithms.

Our experimental results indicate that careful handling of complex Scala features greatly improves call graph precision. We also found that call graphs constructed from the JVM bytecodes using the RTA algorithm are much less precise than those constructed using our source-based algorithms, because significant type information is lost due to the transformations and optimizations performed by the Scala compiler.

In summary, this paper makes the following contributions:

1. We present variations on the RA [26] and RTA [6] algorithms for Scala. To our knowledge, these are the first call graph construction algorithms designed for Scala.
2. We evaluate these algorithms, comparing their relative cost and precision on a set of publicly available Scala programs.
3. We evaluate the application of the RTA algorithm to the JVM bytecodes produced by the Scala compiler, and show that such an approach is not viable because it produces highly imprecise call graphs.

In addition, we have formalized our most precise algorithm and proven its correctness in a separate technical report [4].

The remainder of this paper is organized as follows. Section 2 reviews existing call graph construction algorithms that serve as the inspiration for our work. Section 3 presents a number of motivating examples that illustrate the challenges associated with constructing call graphs of Scala programs. Section 4 presents our algorithms. Section 5 presents the implementation in the context of the Scala compiler. An evaluation of our algorithms is presented in Section 6. Lastly, Section 7 concludes and briefly discusses directions for future work.

2 Background

Algorithms for call graph construction [11] have been studied extensively in the context of object-oriented programming languages such as Java [10, 17], C++ [6] and Self [1], of functional programming languages such as Scheme [24] and ML [12], and of scripting languages such as JavaScript [25]. Roughly speaking, most call graph construction algorithms can be classified as being either *type-based* or *flow-based* [7, 13, 14, 17, 18]. The former class of algorithms uses only local information given by static types to determine possible call targets, whereas the latter analyzes the program's data flow.

We focus on type-based algorithms, so we will briefly review some important type-based call graph construction algorithms for object-oriented languages upon which our work is based. In the exposition of these algorithms, we use a constraint notation that is equivalent to that of [27], but that explicitly represents call graph edges using a relation '$\mapsto$' between call sites and methods.

Name-Based Resolution (RA). The main challenge in constructing call graphs of object-oriented programs is in approximating the behavior of dynamically dispatched (virtual) method calls. Early work (see, e.g., [26]) simply assumed that a virtual call $e.m(\cdots)$ can invoke any method with the same name m. This approach can be captured using the following constraints:

$$\frac{}{\mathit{main} \in R}\ \text{RA}_{\text{MAIN}}$$

$$\frac{c \mapsto M}{M \in R}\ \text{RA}_{\text{REACHABLE}} \qquad \frac{\begin{array}{c}\text{call } c : e.m(\ldots) \text{ occurs in method } M \\ \text{method } M' \text{ has name } m \\ M \in R\end{array}}{c \mapsto M'}\ \text{RA}_{\text{CALL}}$$

Intuitively, rule RA_{MAIN} reads "the main method is reachable" by including it in the set R of reachable methods. Rule RA_{CALL} states that "if a method is reachable, and a call site $c : e.m(\ldots)$ occurs in its body, then every method with name m is reachable from c." Finally, rule $\text{RA}_{\text{REACHABLE}}$ states that any method M reachable from a call site c is contained in the set R of reachable methods.

Class Hierarchy Analysis (CHA). Obviously, Name-Based Resolution can become very imprecise if a class hierarchy contains unrelated methods that happen to have the same name. Class Hierarchy Analysis [9] improves upon name-based resolution by using the static type of the receiver expression of a method call in combination with class hierarchy information to determine what methods may be invoked from a call site. Following the notation of [27], we use $\mathit{StaticType}(e)$ to denote the static type of an expression e, and $\mathit{StaticLookup}(C, m)$ to denote the method definition that is invoked when method m is invoked on an object with run-time type C. Using these definitions, CHA is defined as follows:

$$\frac{}{\mathit{main} \in R}\ \text{CHA}_{\text{MAIN}}$$

$$\frac{c \mapsto M}{M \in R}\ \text{CHA}_{\text{REACHABLE}} \qquad \frac{\begin{array}{c}\text{call } c : e.m(\ldots) \text{ occurs in method } M \\ C \in \mathit{SubTypes}(\mathit{StaticType}(e)) \\ \mathit{StaticLookup}(C, m) = M' \\ M \in R\end{array}}{c \mapsto M'}\ \text{CHA}_{\text{CALL}}$$

Rules CHA$_{\text{MAIN}}$ and CHA$_{\text{REACHABLE}}$ are the same as their counterparts for RA. Intuitively, rule CHA$_{\text{CALL}}$ now reads: “if a method is reachable, and a call site $c : e.m(\ldots)$ occurs in the body of that method, then every method with name m that is inherited by a subtype of the static type of e is reachable from c.”

Rapid Type Analysis (RTA). Bacon and Sweeney [5,6] observed that CHA produces very imprecise results when only a subset of the classes in an application is instantiated. In such cases, CHA loses precision because, effectively, it assumes for a method call $e.m(\cdots)$ that all subtypes of the static type of e may arise at run time. In order to mitigate this loss of precision, RTA maintains a set of types $\hat{\Sigma}$ that have been instantiated in reachable methods. This set is used to approximate the types that a receiver expression may assume at run time. The constraint formulation of RTA is as follows:

$$\frac{}{\mathit{main} \in R}\ \text{RTA}_{\text{MAIN}}$$

$$\frac{\begin{array}{c}\text{“new } C()\text{” occurs in } M \\ M \in R\end{array}}{C \in \hat{\Sigma}}\ \text{RTA}_{\text{NEW}}$$

$$\frac{c \mapsto M}{M \in R}\ \text{RTA}_{\text{REACHABLE}}$$

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ C \in \mathit{SubTypes}(\mathit{StaticType}(e)) \\ \mathit{StaticLookup}(C, m) = M' \\ M \in R \\ C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{RTA}_{\text{CALL}}$$

Rules RTA$_{\text{MAIN}}$ and RTA$_{\text{REACHABLE}}$ are again the same as before. Intuitively, RTA$_{\text{CALL}}$ refines CHA$_{\text{CALL}}$ by requiring that $C \in \hat{\Sigma}$, and rule RTA$_{\text{NEW}}$ reads: “$\hat{\Sigma}$ contains the classes that are instantiated in a reachable method.”

Sallenave and Ducourneau [22] recently presented an extension of RTA for the C# language that determines the types with which parameterized classes are instantiated by maintaining sets of type tuples for parameterized classes and methods. They use their analysis to generate efficient CLI code for embedded applications that avoids expensive boxing/unboxing operations on primitive types, while permitting a space-efficient shared representation for reference types.

3 Motivating Examples

Before presenting our algorithms in Section 4, we briefly review the Scala features that pose the most significant challenges for call graph construction.

3.1 Traits

Traits are one of the cornerstone features of Scala. They provide a flexible mechanism for distributing the functionality of an object over multiple reusable components. Traits are similar to Java’s abstract classes in the sense that they may provide definitions of methods, and in that they cannot be instantiated by themselves. However, they resemble Java interfaces in the sense that a trait may extend (“mix-in”) multiple super-traits.

```
 1 object Traits {
 2   trait A {
 3     def foo = println("A.foo")
 4     def bar
 5   }
 6   trait B {
 7     def foo
 8     def bar = this.foo
 9   }
10   trait C {
11     def foo = println("C.foo")
12   }
13
14   def main(args: Array[String]) =
15     { (new A with B).bar }
16 }
```

Fig. 1. A Scala program illustrating the use of traits.

Figure 1 shows an example program that declares a trait A in which a concrete method foo and an abstract method bar are defined. The program also declares a trait B that defines a concrete method bar and an abstract method foo. Lastly, trait C defines a concrete method foo. The program contains a main method that creates an object by composing A and B, and then calls bar on that object.

Before turning our attention to call graph construction, we need to consider how method calls are resolved in Scala. In Scala, the behavior of method calls depends on the class linearization order of the receiver object [19, Section 5.1.2]. The *linearization* of a class C with parents C_1 with $\cdots$ with C_n is defined as:

$$\mathcal{L}(C) = C, \mathcal{L}(C_n) \vec{+} \cdots \vec{+} \mathcal{L}(C_1)$$

where $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left one[3]. Scala defines the set of *members* of a class in terms of its linearization. Ignoring a number of complicating factors detailed in the Scala specification [19, §5.1.3 and §5.1.4], the members of a class C include all members m declared in classes in $\mathcal{L}(C)$, except for those overridden in classes that precede C in the linearization order. Given this notion of class membership, the resolution of method calls is straightforward: a call $x.m(\cdots)$ where x has type C at run time dispatches to the unique member named m in C.

For the example of Figure 1, the linearization order of type new A with B on line 15 is: X, B, A (here, we use X to denote the anonymous class that is implicitly declared by the allocation expression new A with B). Following the definitions above, the set of members of X is: { B.bar, A.foo }. Hence, the call to bar on line 15 resolves to B.bar. Using a similar argument, the call to foo on line 8 resolves to A.foo. Therefore, executing the program will print “A.foo”.

Implications for call graph construction. The presence of traits complicates the construction of call graphs because method calls that occur in a trait typically cannot be resolved by consulting the class hierarchy alone. In the example of Figure 1, B.bar contains a call this.foo on line 8. How should a call graph construction algorithm approximate the behavior of this call, given that there is no inheritance relation between A, B, and C?

[3] The presence of an allocation expression such as new C with D is equivalent to a declaration of a new empty class with parents C with D.

To reason about the behavior of method calls in traits, a call graph construction algorithm needs to make certain assumptions about how traits are combined. One very conservative approach would be to assume that a program may combine each trait with any set of other traits in the program in any order[4], such that the resulting combination is syntactically correct[5]. Then, for each of these combinations, one could compute the members contained in the resulting type, and approximate the behavior of calls by determining the method that is selected in each case. For the program of Figure 1, this approach would assume that B is composed with either A or with C. In the former case, the call on line 8 is assumed to be invoked on an object of type A with B (or B with A), and would dispatch to A.foo. In the latter, the call is assumed to be invoked on an object of type C with B (or B with C), and would dispatch to C.foo. Hence, a call graph would result in which both A.foo and C.foo are reachable from the call on line 8.

The conservative approach discussed above is likely to be imprecise and inefficient in cases where a program contains many traits that can be composed with each other. For practical purposes, a better approach is to determine the set of combinations of traits that actually occur in the program, and to use that set of combinations of traits to resolve method calls. Returning to our example program, we observe that the only combination of traits is A with B, on line 15. If the call on line 8 is dispatched on an object of this type, it will dispatch to A.foo, as previously discussed. Hence, this approach would create a smaller call graph in which there is only one outgoing edge for the call on line 8.

This more precise approach requires that the set of all combinations of traits in the program can be determined. The conservative approach could still have merit in cases where this information is not available (e.g., libraries intended to be extended with code that instantiates additional trait combinations).

3.2 Abstract Type Members

Scala supports a flexible mechanism for declaring *abstract type members* in traits and classes. A *type declaration* [19, §4.3] defines a name for an abstract type, along with upper and lower bounds that impose constraints on the concrete types that it could be bound to. An abstract type is bound to a concrete type when its declaring trait is composed with (or extended by) another trait that provides a concrete definition in one of two ways: either it contains a class or trait with the same name as the abstract type, or it declares a *type alias* [19, §4.3] that explicitly binds the abstract type to some specified concrete type.

Figure 2 shows a program that declares traits X, Y, Z, and HasFoo. Traits X and Y each declare a member class A that is a subclass of HasFoo. Traits Y and Z each declare an abstract type member B and a field o, which is assigned a new

[4] Note that an X with Y object may behave differently from a Y with X object in certain situations because these objects have different linearization orders.

[5] If multiple traits that provide concrete definitions of the same method are composed, all but the last of these definitions in the linearization order must have the override modifier in order for the composition to be syntactically correct [19, Section 5.1.4].

```
17 object AbstractTypeMembers {
18   trait HasFoo {
19     def foo: Unit
20   }
21   trait X {
22     class A extends HasFoo {
23       def foo = println("X.A.foo")
24     }
25   }
26   trait Y {
27     class A extends HasFoo {
28       def foo = println("Y.A.foo")
29     }
30     ...
31     ...
32     type B = A
33     val o = new A
34   }
35   trait Z {
36     type B <: HasFoo
37     val o: B
38     def bar = o.foo
39   }
40
41   def main(args: Array[String]) =
42     { (new Y with Z {}).bar }
43 }
```

Fig. 2. A Scala program illustrating the use of abstract type members.

A object in Y. Note that Y defines its B to be the same as Y.A. Observe that the abstract member type B of Z has a bound HasFoo, and that o is declared to be of type B. The presence of this bound means that we can call foo on o on line 38.

On line 42, the program creates an object by composing Y with Z, and calls bar on it. Following Scala's semantics for method calls, this call will dispatch to Z.bar. To understand how the call o.foo on line 38 is resolved, we must understand how abstract type members are bound to concrete types as a result of trait composition. In this case, the composition of Y with Z means that the types Y.B and Z.B are unified. Since Y.B was defined to be the same as Y.A, it follows that the abstract type member Z.B is bound to the concrete type Y.A. Thus, executing the call on line 38 dispatches to Y.A.foo, so the program prints "Y.A.foo".

Implications for call graph construction. How could a call graph construction algorithm approximate the behavior of calls such as o.foo in Figure 2, where the receiver expression's type is abstract? A conservative solution relies on the bound of the abstract type as follows: For a call $o.f(\cdots)$ where o is of an abstract type T with bound B, one could assume the call to dispatch to definitions of $f(\cdots)$ in any subtype of B. This approach is implemented in our TCA^{bounds} algorithm and identifies both X.A.foo and Y.A.foo as possible targets of the call on line 38.

However, the above approach may be imprecise if certain subtypes of the bound are not instantiated. Our TCA^{expand} algorithm implements a more precise approach that considers how abstract type members are bound to concrete types in observed combinations of traits, in the same spirit of the more precise treatment of trait composition discussed above. In Figure 2, the program only creates an object of type Y with Z, and Z.B is bound to Y.A in this particular combination of traits. Therefore, the call on line 38 must dispatch to Y.A.foo.

Scala's parameterized types [19, §3.2.4] resemble abstract type members and are handled similarly. Similar issues arise in other languages with generics [22].

```
44 object Closures {
45   def bar1(y: () => A) = { y() }
46   def bar2(z: () => B) = { z() }
47
48   class A
49   class B
50   def main(args: Array[String]) = {
51     val foo1 = () => { new A }
52     val foo2 = () => { new B }
53     this.bar1(foo1)
54     this.bar2(foo2)
55   }
56 }
```

Fig. 3. A Scala program illustrating the use of closures.

3.3 Closures

Scala allows functions to be bound to variables and passed as arguments to other functions. Figure 3 illustrates this feature, commonly known as "closures". On line 51, the program creates a function and assigns it to a variable foo1. The function's declared type is () => A, indicating that it takes no parameters and returns an object of type A. Likewise, line 52 assigns to foo2 a function that takes no arguments and returns a B object.

Next, on line 53, bar1 is called with foo1 as an argument. Method bar1 (line 45) binds this closure to its parameter y, which has declared type () => A, and then calls the function bound to y. Similarly, on line 54 bar2 is called with foo2 as an argument. On line 46, this closure is bound to a parameter z and then invoked. From the simple data flow in this example, it is easy to see that the call y() on line 45 always calls the function that was bound to foo1 on line 51, and that the call z() on line 46 always calls the function that was bound to foo2 on line 52.

Implications for call graph construction. In principle, one could use the declared types of function-valued expressions and the types of the closures that have been created to determine if a given call site could invoke a given function. For example, the type of y is () => A, and line 53 creates a closure that can be bound to a variable of this type. Therefore, a call graph edge needs to be constructed from the call site y() to the closure on line 53. By the same reasoning, a call graph edge should be constructed from the call site z() to the closure on line 54.

Our implementation takes a different approach to handle closures. Rather than performing the analysis at the source level, we apply it after the Scala compiler has "desugared" the code by transforming closures into anonymous classes that extend the appropriate scala.runtime.AbstractFunction$N$. Each such class has an apply() method containing the closure's original code. Figure 4 shows a desugared version of the program of Figure 3. After this transformation, closures can be treated as ordinary parameterized Scala classes without loss of precision. This has the advantage of keeping our implementation simple and uniform.

3.4 Calls on the variable this

Figure 5 shows a program that declares a trait A with subclasses B and C. Trait A declares an abstract method foo, which is overridden in B and C, and a concrete

```
object Closures {
  def bar1(y: () => A) = { y.apply() }
  def bar2(z: () => B) = { z.apply() }

  class A
  class B

  def main(args: Array[String]) = {
    val foo1: () => A = {
      class $anonfun extends
        scala.runtime.AbstractFunction0[A] {
          def apply(): A = { new A() }
        };
        new $anonfun()
    };
    val foo2: () => B = {
      class $anonfun extends
        scala.runtime.
          AbstractFunction0[B] {
          def apply(): B = {
            new B()
          }
       };
       new $anonfun()
    };
    this.bar1(foo1)
    this.bar2(foo2)
  }
}
```

Fig. 4. Desugared version of the program of Figure 3 (slightly simplified).

```
86  object This {
87    trait A {
88      def foo
89      // can only call B.foo
90      def bar = this.foo
91    }
92
93    class B extends A {
94      def foo = println("B.foo")
95    }
96    class C extends A {
97      def foo = println("C.foo")
98      override def bar = println("C.bar")
99    }
100
101   def main(args: Array[String]) = {
102     (new B).bar
103     (new C).bar
104   }
105 }
```

Fig. 5. A Scala program illustrating a call on this.

method bar, which is overridden in C (but not in B). The program declares a main method that calls bar on objects of type B and C (lines 102–103). Executing the call to bar on line 102 dispatches to A.bar(). Executing the call this.foo() in that method will then dispatch to B.foo(). Finally, executing the call to bar on line 103 dispatches to C.bar, so the program prints “B.foo”, then “C.bar”.

Consider how a call graph construction algorithm would approximate the behavior of the call this.foo() at line 90. The receiver expression’s type is A, so CHA concludes that either B.foo or C.foo could be invoked, since B and C are subtypes of A. However, note that this cannot have type C in A.bar because C provides an overriding definition of bar. Stated informally, this cannot have type C inside A.bar because then execution would not have arrived in A.bar in the first place. The TCA$^{expand\text{-}this}$ algorithm, presented in Section 4, exploits such knowledge. Care must be taken in the presence of super-calls, as we will discuss.

3.5 Bytecode-based Analysis

The above examples show that Scala's traits and abstract type members pose new challenges for call graph construction. Several other Scala features, such as path-dependent types and structural types, introduce further complications, and will be discussed in Section 5. At this point, the reader may wonder if all these complications could be avoided by simply analyzing the JVM bytecodes produced by the Scala compiler.

We experimentally determined that such an approach is not viable for two reasons. First, the translation of Scala source code to JVM bytecode involves significant code transformations that result in the loss of type information, causing the computed call graphs to become imprecise. Second, the Scala compiler generates code containing hard-to analyze reflection for certain Scala idioms.

Loss of Precision. Consider Figure 6, which shows JVM bytecode produced by the Scala compiler for the program of Figure 3. As can be seen in the figure, the closures that were defined on lines 51 and 52 in Figure 3 have been translated into classes Closures$$anonfun$1 (lines 128–138 in Figure 6) and Closures$$anonfun$2 (lines 140–150). These classes extend scala.runtime.AbstractFunction0<T>, which is used for representing closures with no parameters at the bytecode level. Additionally, these classes provide overriding definitions for the apply method inherited by scala.runtime.AbstractFunction0<T> from its super-class scala.Function0<T>. This apply method returns an object of type T. The issue to note here is that Closures$$anonfun$1 and Closures$$anonfun$2 each instantiate the type parameter T with different types, Closures$A and Closures$B, respectively. Therefore, their apply methods return objects of type Closures$A and Closures$B. However, at the bytecode level, all type parameters are erased, so that we have a situation where:

- scala.Function0.apply has return type Object
- Closures$$anonfun$1.apply and Closures$$anonfun$2.apply each override scala.Function0.apply and also have return type Object
- there are two calls to scala.Function0.apply on lines 118 and 123

Given this situation, the RTA algorithm creates edges to Closures$$anonfun$1.apply and Closures$$anonfun$2.apply from each of the calls on lines 118 and 123. In other words, a bytecode-based RTA analysis creates 4 call graph edges for the closure-related calls, whereas the analysis of Section 3.3 only created 2 edges. In Section 6, we show that this scenario commonly arises in practice, causing bytecode-based call graphs to become extremely imprecise.

Reflection in Generated Code. We detected several cases where the Scala compiler generates code that invokes methods using java.lang.reflect.Method.invoke(). In general, the use of reflection creates significant problems for static analysis, because it must either make very conservative assumptions that have a detrimental effect on precision (e.g., assuming that calls to java.lang.reflect.Method.invoke() may invoke any method in the application) or the analysis will become unsound.

Figure 7 shows a small example (taken from the ENSIME program, see Section 6) for which the Scala compiler generates code containing reflection.

```
public final class Closures$ {
  public void main(java.lang.String []);
     0: new Closures$$anonfun$1
      ...
     8: new Closures$$anonfun$2
      ...
    18: invokevirtual Closures$.bar1(scala.Function0) : void
      ...
    23: invokevirtual Closures$.bar2(scala.Function0) : void
    26: return
  public void bar1(scala.Function0);
     0: aload_1
     1: invokeinterface scala.Function0.apply() : java.lang.Object
     6: pop
     7: return
  public void bar2(scala.Function0);
     0: aload_1
     1: invokeinterface scala.Function0.apply() : java.lang.Object
     6: pop
     7: return
}

public final class Closures$$anonfun$1 extends scala.runtime.AbstractFunction0 {
   public final Closures$A apply();
     0: new Closures$A
     3: dup
     4: invokespecial Closures$A()
     7: areturn
  public final java.lang.Object apply();
     0: aload_0
     1: invokevirtual Closures$$anonfun$1.apply() : Closures$A
     4: areturn
}

public final class Closures$$anonfun$2 extends scala.runtime.AbstractFunction0 {
   public final Closures$B apply();
     0: new Closures$B
     3: dup
     4: invokespecial Closures$B()
     7: areturn
  public final java.lang.Object apply();
     0: aload_0
     1: invokevirtual Closures$$anonfun$2.apply() : Closures$B
     4: areturn
}
```

Fig. 6. JVM bytecode produced by the Scala compiler for the program of Figure 3.

```
trait ClassHandler

object LuceneIndex {
  def buildStaticIndex(): Int = {
    val handler = new ClassHandler {
      var classCount = 0
      var methodCount = 0
    }
    handler.classCount + handler.methodCount
  }
}
```

Fig. 7. A Scala program for which the compiler generates code containing reflective method calls (taken from the ENSIME program, see Section 6).

4 Algorithms

We present a family of call graph construction algorithms using generic inference rules, in the same style that we used in Section 2. The algorithms presented here are: TCA^{names}, a variant of RA that considers only types instantiated in reachable code, TCA^{bounds}, a variant of RTA adapted to deal with Scala's trait composition and abstract type members, TCA^{expand}, which handles abstract type members more precisely, and TCA$^{expand\text{-}this}$, which is more precise for call sites where the receiver is this.

We use Figure 8 to illustrate differences between the algorithms. When executed, the call site on line 172 calls method B.foo; our different algorithms resolve this call site to various subsets of the foo methods in classes A, B, C, and D.

```
162 class A { def foo = "A.foo" }
163 class B extends A { override def foo = "B.foo" }
164 class C { def foo = "C.foo" }
165 class D { def foo = "D.foo" }
166 class CallSiteClass [T <: A](val receiver : T) {
167   def callsite = {
168     /* resolves to:
169      *   TCA^expand: { B.foo }, TCA^bounds: { B.foo, A.foo }
170      *   TCA^names: { B.foo, A.foo, C.foo } ,   RA: { B.foo, A.foo, C.foo, D.foo }
171      */
172     receiver .foo
173   }
174 }
175 def main(args: Array[String ]): Unit = {
176   new A
177   val receiver = new B
178   new C
179   val callSiteClass = new CallSiteClass[B]( receiver );
180    callSiteClass . callsite
181 }
```

Fig. 8. A Scala program illustrating the varying precision of the analyses.

4.1 TCA^{names}

The RA algorithm of Section 2 is sound for Scala because it resolves calls based only on method names, and makes no use of types. However, it is imprecise because it considers as possible call targets all methods that have the appropriate name, even those in unreachable code. For Figure 8, RA resolves the call site as possibly calling all four foo methods, even though D is never instantiated in code reachable from main. Since RA already computes a set R of reachable methods, we extend it to consider only classes and traits instantiated in reachable methods.

We add rule RTA_{NEW} from RTA, which computes a set $\hat{\Sigma}$ of types instantiated in reachable methods. The CALL rule[6] is adapted as follows to make use of $\hat{\Sigma}$:

$$\frac{\begin{array}{c}\text{call } c : e.m(\ldots) \text{ occurs in method } M \\ \text{method } M' \text{ has name } m \\ \colorbox{lightgray}{method M' is a member of type C} \\ M \in R \qquad \colorbox{lightgray}{$C \in \hat{\Sigma}$}\end{array}}{c \mapsto M'}\ \text{TCA}^{names}_{\text{CALL}}$$

The resulting TCA^{names} analysis consists of the rule RTA_{NEW} and the rules of RA, except that RA_{CALL} is replaced with $\text{TCA}^{names}_{\text{CALL}}$. In $\text{TCA}^{names}_{\text{CALL}}$, a method is considered as a possible call target only if it is a member of some type C that has been instantiated in a reachable method in R[7].

For the program of Figure 8, TCA^{names} resolves the call site to A.foo, B.foo, and C.foo, but not D.foo because D is never instantiated in reachable code.

4.2 TCA^{bounds}

To improve precision, analyses such as RTA and CHA use the static type of the receiver e to restrict its possible runtime types. Specifically, the runtime type C of the receiver of the call must be a subtype of the static type of e.

A key difficulty when analyzing a language with traits is enumerating all subtypes of a type, as both CHA and RTA do in the condition $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ in rules CHA_{CALL} and RTA_{CALL} of Section 2. Given a trait T, any composition of traits containing T is a subtype of T. Therefore, enumerating possible subtypes of T requires enumerating all compositions of traits. Since a trait composition is an ordered list of traits, the number of possible compositions is exponential in the number of traits[8].

In principle, an analysis could make the conservative assumption that all compositions of traits are possible, and therefore that any method defined in any trait can override any other method of the same name and signature in any other trait (a concrete method overrides another method with the same name and signature occurring later in the linearization of a trait composition). The resulting analysis would have the same precision as the name-based algorithms RA and TCA^{names}, though it would obviously be much less efficient.

Therefore, we consider only combinations of traits occurring in reachable methods of the program. This set of combinations is used to approximate the

[6] When we present an inference rule in this section, we use shading to highlight which parts of the rule are modified relative to similar preceding rules.

[7] Calls on super require special handling, as will be discussed in Section 5.

[8] Although some trait compositions violate the well-formedness rules of Scala, such violations are unlikely to substantially reduce the exponential number of possible compositions. Moreover, the well-formedness rules are defined in terms of the members of a specific composition, so it would be difficult to enumerate only well-formed compositions without first examining all of them.

behavior of method calls. In essence, this is similar to the closed-world assumption of RTA. Specifically, the TCA^{bounds} analysis includes the rule RTA$_{\text{NEW}}$ to collect the set $\hat{\Sigma}$ of trait combinations occurring at reachable allocation sites. The resulting set is used in the following call resolution rule:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M\\ C \in \mathit{SubTypes}(\mathit{StaticType}(e))\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{bounds}_{\text{CALL}}$$

The added check $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ relies on the subtyping relation defined in the Scala language specification, which correctly handles complexities of the Scala type system such as path-dependent types.

According to Scala's definition of subtyping, abstract types do not have subtypes, so TCA$^{bounds}_{\text{CALL}}$ does not apply. Such a definition of subtyping is necessary because it cannot be determined locally, just from the abstract type, which actual types will be bound to it elsewhere in the program. However, every abstract type in Scala has an upper bound (if it is not specified explicitly, scala.Any is assumed), so an abstract type T can be approximated using its upper bound B:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M\\ \mathit{StaticType}(e) \text{ is an abstract type with upper bound } B\\ C \in \mathit{SubTypes}(\,B\,)\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{bounds}_{\text{ABSTRACT-CALL}}$$

For the program of Figure 8, TCA^{bounds} resolves the call site to A.foo and B.foo, but not D.foo because D is never instantiated, and not C.foo, because C is not a subtype of A, the upper bound of the static type T of the receiver.

4.3 TCA^{expand}

The TCA^{bounds} analysis is particularly imprecise for abstract types that do not have a declared upper bound, since using the default upper bound of scala.Any makes the bound-based analysis as imprecise as the name-based analysis.

It is more precise to consider only concrete types with which each abstract type is instantiated, similar to the approach of [22]. To this end, we introduce a mapping *expand*(), which maps each abstract type[9] T to those concrete types with which it has been instantiated:

[9] Similar rules (not shown) are needed to handle the type parameters of generic types and type-parametric methods. Our implementation fully supports these cases.

$$\dfrac{\begin{array}{c} C \in \hat{\Sigma} \\ \text{“type } A = B\text{” is a member of } C \\ D \text{ is a supertype of } C \\ \text{“type } A\text{” is a member of } D \end{array}}{B \in \mathit{expand}(D.A)}\ \text{TCA}^{\mathit{expand}}_{\text{EXPAND-TYPE}}$$

$$\dfrac{\begin{array}{c} C \in \hat{\Sigma} \\ \text{“trait } A\ \{\ \ldots\}\text{” is a member of } C \\ D \text{ is a supertype of } C \\ \text{“type } A\text{” is a member of } D \end{array}}{C.A \in \mathit{expand}(D.A)}\ \text{TCA}^{\mathit{expand}}_{\text{EXPAND-TRAIT}}$$

$$\dfrac{\begin{array}{c} R \in \mathit{expand}(S) \\ S \in \mathit{expand}(T) \end{array}}{R \in \mathit{expand}(T)}\ \text{TCA}^{\mathit{expand}}_{\text{EXPAND-TRANS}}$$

The $\text{TCA}^{\mathit{bounds}}_{\text{CALL}}$ rule is then updated to use the *expand*() mapping to determine the concrete types bound to the abstract type of a receiver:

$$\dfrac{\begin{array}{c} \text{call } e.m(\ldots) \text{ occurs in method } M \\ \mathit{StaticType}(e) \text{ is an abstract type } T \\ C \in \mathit{SubTypes}(\ \colorbox{lightgray}{$\mathit{expand}(T)$}\ ) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma} \end{array}}{c \mapsto M'}\ \text{TCA}^{\mathit{expand}}_{\text{ABSTRACT-CALL}}$$

Rule $\text{TCA}^{\mathit{expand}}_{\text{EXPAND-TYPE}}$ handles situations such as the one where a type assignment type A = B is a member of some instantiated trait composition C. Now, if a supertype D of C declares an abstract type A, then B is a possible concrete instantiation of the abstract type D.A, and this fact is recorded in the *expand*() mapping by $\text{TCA}^{\mathit{expand}}_{\text{EXPAND-TYPE}}$. Rule $\text{TCA}^{\mathit{expand}}_{\text{EXPAND-TRAIT}}$ handles a similar case where an abstract type is instantiated by defining a member trait with the same name. The right-hand-side of a type assignment might be abstract, so it is necessary to compute the transitive closure of the *expand*() mapping (rule $\text{TCA}^{\mathit{expand}}_{\text{EXPAND-TRANS}}$).

Cycles among type assignments may exist. In Scala, cyclic references between abstract type members are a compile-time error. However, recursion in generic types is allowed. For example, the parameter B in a generic type A[B] could be instantiated with A[B] itself, leading to B representing an unbounded sequence of types A[B], A[A[B]], ... This kind of recursion can be detected either by limiting the size of $\mathit{expand}(T)$ for each abstract type to some fixed bound, or by checking for occurrences of T in the expansion $\mathit{expand}(T)$. The current version of our implementation limits the size of $\mathit{expand}(T)$ to 1000 types. This bound was never exceeded in our experimental evaluation, implying that recursive types did not occur in the benchmark programs. The same issue also occurs in Java and C#, and was previously noted by Sallenave and Ducourneau [22]. Their implementation issues a warning when it detects the situation. Our algorithm resolves the issue soundly: when a recursive type T is detected, the algorithm falls back to using the upper bound of T to resolve calls on receivers of type T.

4.4 TCA$^{\textit{expand-this}}$

In both Java and Scala, calls on the this reference are common. In some cases, it is possible to resolve such calls more precisely by exploiting the knowledge that the caller and the callee must be members of the same object. Care must be taken in the presence of super-calls, as will be discussed in Section 5.1.

For example, at the call this.foo() on line 90 of Figure 5, the static type of the receiver this is A, which has both B and C as subtypes. Since B and C are both instantiated, all of the analyses described so far would resolve the call to both B.foo (line 94) and C.foo (line 97). However, any object that has C.foo as a member also has C.bar as a member, which overrides the method A.bar containing the call site. Therefore, the call site at line 90 can never resolve to method C.foo.

This pattern is handled precisely by the following rule:

$$\frac{\begin{array}{c} \text{call } D.this.m(\ldots) \text{ occurs in method } M \\ D \text{ is the declaring trait of } M \\ C \in \mathit{SubTypes}(D) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ \text{method } M \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma} \end{array}}{c \mapsto M'} \ \text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$$

The rule requires not only the callee M', but also the caller M to be members of the same instantiated type C. The rule applies only when the receiver is the special variable this. Because nested classes and traits are common in Scala, it is possible that a particular occurrence of the special variable this is qualified to refer to the enclosing object of some outer trait. Since it would be unsound to apply $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$ in this case, we require that the receiver be the special variable this of the innermost trait D that declares the caller method M.

After adding rule $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$, we add a precondition to rule $\text{TCA}^{\textit{expand-this}}_{\text{CALL}}$ so that it does not apply when $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$ should, i.e., when the receiver is the special variable this of the declaring trait D of the caller method M.

4.5 Correctness

In a separate technical report [4], we provide a formalization of the inference rules for TCA$^{\textit{expand-this}}$ based on the FS_{alg} ("Featherweight Scala") representation of Cremet et al. [8]. We also prove the TCA$^{\textit{expand-this}}$ analysis correct with respect to the operational semantics of FS_{alg} by demonstrating that:

1. For any FS_{alg} program P, the set of methods called in an execution trace of P is a subset of the set R of reachable methods computed for P by TCA$^{\textit{expand-this}}$.
2. For any FS_{alg} program P, if the execution trace of P contains a call from call site c to a target method M, then TCA$^{\textit{expand-this}}$ applied to P derives $c \mapsto M$.

5 Implementation

We implemented RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, and $\text{TCA}^{expand\text{-}this}$ as a plugin for version 2.10.2 of the Scala compiler, and tested the implementation on a suite of programs exhibiting a wide range of Scala features. To the best of our knowledge, our analyses soundly handle the entire Scala language, but we assume that all code to be analyzed is available and we ignore reflection and dynamic code generation. We also used the implementation of RTA in the WALA framework to construct call graphs from JVM bytecode.

The analysis runs after the `uncurry` phase, which is the 12th of 30 compiler phases. At this stage, most of the convenience features in Scala that are specified as syntactic sugar have been desugared. However, the compiler has not yet transformed the program to be closer to JVM bytecode, and has not yet erased any significant type information. In particular, closures have been turned into function objects with apply methods, pattern matching has been desugared into explicit tests and comparisons, and implicit calls and parameters have been made explicit, so our analysis does not have to deal with these features explicitly.

Some Scala idioms, e.g., path-dependent types, structural types, singletons, and generics, make the subtype testing in Scala complicated [19, §3.5]. Fortunately, we can rely on the Scala compiler infrastructure to answer subtype queries. Two issues, however, require special handling in the implementation: super calls and incomplete programs.

5.1 Super calls

Normally, when a method is called on some receiver object, the method is a member of that object. Super calls violate this general rule: a call on super invokes a method in a supertype of the receiver's type. This method is generally not a member of the receiver object, because some other method overrides it.

At a call on super, the analysis must determine the method actually invoked. When the call site is in a class (not a trait), the call is resolved statically as in Java. When the call site is in a trait, however, the target method is selected using a dynamic dispatch mechanism depending on the runtime type of the receiver [19, §6.5]. Our analysis resolves such calls using a similar procedure as for ordinary dynamically dispatched calls. For each possible run-time type of the receiver, the specified procedure is followed to find the actual call target.

The $\text{TCA}^{expand\text{-}this}$ analysis requires that within any method M, the this variable refers to an object of which M is a member. This premise is violated when M is invoked using a super call. To restore soundness, we blacklist the signatures of the targets of all reachable super calls. Within a method whose signature is blacklisted, we fall back to the TCA^{expand} analysis instead of $\text{TCA}^{expand\text{-}this}$.

5.2 Incomplete programs

Our analyses are defined for complete programs, but a practical implementation must deal with incomplete programs. A typical example of an incomplete program is a situation where user code calls unanalyzed libraries.

Our implementation analyzes Scala source files presented to the compiler, but not referenced classes provided only as bytecode such as the Scala and Java standard libraries. The analysis soundly analyzes call sites occurring in the provided Scala source files using a Scala analogue of the Separate Compilation Assumption [2,3], which asserts that unanalyzed "library" classes do not directly reference analyzed "application" classes. If application code passes the name of one of its classes to the library and the library instantiates it by reflection, then our analysis faces the same challenges as any Java analysis, and the same solutions would apply.

If the declaring class of the static target of a call site is available for analysis, then so are all its subtypes. In such cases, the analysis can soundly determine all possible actual call targets. On the other hand, if the declaring class of the static target of a call is in an unanalyzed class, it is impossible to determine all possible actual target methods, because some targets may be in unanalyzed code or in trait compositions that are only created in unanalyzed code. The implementation records the existence of such call sites, but does not attempt to resolve them soundly. However, such call sites, as well as those in unanalyzed code, may call methods in analyzed code via call-backs. For soundness, the analysis must treat such target methods as reachable. This is achieved by considering a method reachable if it occurs in an instantiated type and if it overrides a method declared in unanalyzed code. This is sound because in both cases (a call whose static target is in unanalyzed code, or a call in unanalyzed code), the actual runtime target method must override the static target of the call.

Determining the method overriding relationship is more difficult than in Java. Two methods declared in two independent traits do not override each other unless these traits are composed in the instantiation of some object. Therefore, the overriding relation must be updated as new trait compositions are discovered.

6 Evaluation

We evaluated our implementation on publicly available Scala programs covering a range of different application areas and programming styles.[10] Table 1 shows, for each program, the number of lines of Scala source code (excluding library code), classes, objects, traits, trait compositions, methods, closures, call sites, call sites on abstract types, and call sites on the variable `this`. ARGOT is a command-line argument parser for Scala. ENSIME is an Emacs plugin that provides an enhanced Scala interactive mode, including a read-eval-print loop (REPL) and many features commonly found in IDEs such as live error-checking,

[10] The benchmark source code is available from `http://github.com/bmc/argot`, `http://github.com/aemoncannon/ensime`, `http://github.com/KarolS/fimpp`, `http://code.google.com/p/kiama`, `http://github.com/colder/phantm`, `http://github.com/eed3si9n/scalaxb`, `http://github.com/Mononofu/Scalisp`, `http://scee.sourceforge.net`, `http://github.com/max-l/Squeryl`, and `http://github.com/nickknw/arbitrarily-sized-tic-tac-toe`.

package/type browsing, and basic refactorings. FIMPP is an interpreter for an imperative, dynamically-typed language that supports integer arithmetic, console output, dynamically growing arrays, and subroutines. KIAMA is a library for language processing used to compile and execute several small languages. PHANTM is a tool that uses a flow-sensitive static analysis to detect type errors in PHP code [16]. SCALAXB is an XML data-binding tool for Scala. SCALISP is a LISP interpreter written in Scala. SEE is a simple engine for evaluating arithmetic expressions. SQUERYL is a Scala library that provides Object-Relational mapping for SQL databases. TICTACTOE is an implementation of the classic “tic-tac-toe” game with a text-based user-interface. Both KIAMA and SCALAXB are part of the DaCapo Scala Benchmarking project [23]. We did not use the other DaCapo Scala benchmarks as they are not compatible with the latest version of Scala.

We ran all of our experiments on a machine with eight dual-core AMD Opteron 1.4 GHz CPUs (running in 64-bit mode) and capped the available memory for the experiments to 16 GB of RAM.

6.1 Research Questions

Our evaluation aims to answer the following Research Questions:

RQ1. How precise are call graphs constructed for the JVM bytecode produced by the Scala compiler compared to analyzing Scala source code?

RQ2. What is the impact on call graph precision of adopting subtype-based call resolution instead of name-based call resolution?

RQ3. What is the impact on call graph precision of determining the set of concrete types that may be bound to abstract type members instead of using a bounds-based approximation?

Table 1. Various characteristics of our benchmark programs.

	LOC	# classes	# objects	# traits	# trait compositions	# methods	# closures	# call sites	# call sites on abstract types	# call sites on this
ARGOT	1,074	18	4	6	185	485	168	2,543	2	276
ENSIME	7,832	223	172	36	984	4,878	532	19,555	23	3,195
FIMPP	1,089	42	53	5	685	2,060	549	5,880	4	1,159
KIAMA	17,914	801	664	162	5,324	19,172	3,963	69,352	401	16,256
PHANTM	9,319	317	358	13	1,498	7,208	561	36,276	15	6,643
SCALAXB	10,290	324	259	222	3,024	10,503	2,204	47,382	35	7,305
SCALISP	795	20	14	0	125	428	115	2,313	23	293
SEE	4,311	130	151	17	415	2,280	262	9,566	11	1,449
SQUERYL	7,432	255	55	110	1,040	3,793	826	13,585	173	2,540
TICTACTOE	247	2	7	0	32	112	24	603	0	41

Table 2. Number of nodes and edges in the summarized version of call graphs computed using the RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, TCA$^{expand\text{-}this}$, and RTA^{wala}.

		RA	**TCA**^{names}	**TCA**^{bounds}	**TCA**^{expand}	**TCA**$^{expand\text{-}this}$	**RTA**^{wala}
ARGOT	nodes	265	184	161	161	161	236
	edges	3,516	1,538	442	442	440	648
ENSIME	nodes	3,491	3,018	2,967	2,966	2,965	4,525
	edges	191,435	150,974	8,025	8,023	8,017	61,803
FIMPP	nodes	870	773	771	771	771	1,381
	edges	12,716	10,900	2,404	2,404	2,404	8,327
KIAMA	nodes	11,959	8,684	7,609	7,600	7,200	13,597
	edges	1,555,533	845,120	35,288	34,062	32,494	609,255
PHANTM	nodes	5,945	5,207	4,798	4,587	4,587	5,157
	edges	376,065	296,252	14,727	13,899	13,870	213,264
SCALAXB	nodes	6,795	2,263	1,196	1,196	1,196	3,866
	edges	1,832,473	322,499	5,819	5,819	5,818	48,966
SCALISP	nodes	283	196	186	186	186	307
	edges	3,807	2,380	526	526	526	908
SEE	nodes	1,869	1,711	1,645	1,572	1,572	2,016
	edges	77,303	63,706	8,349	7,466	7,418	14,520
SQUERYL	nodes	2,484	1,488	408	408	408	1,507
	edges	91,342	46,160	1,677	1,677	1,676	8,669
TICTACTOE	nodes	79	78	78	78	78	112
	edges	524	523	170	170	170	327

RQ4. What is the impact of the special treatment of calls on `this`?
RQ5. How does the running time of the analyses compare?
RQ6. For how many call sites can the algorithms find a single outgoing edge?

6.2 Results

Table 2 summarizes the precision of the call graphs computed by our analyses. For each benchmark and analysis combination, the table shows the number of reachable methods and call edges in the call graph. All call graphs presented in this section include only the analyzed code of the benchmark itself, excluding any library code. For RTA^{wala}, such "summarized call graphs" were obtained by collapsing the parts of the call graph in the library into a single node.

RQ1. To answer this question, we compare the call graphs from the TCA^{bounds} and RTA^{wala} analyses. The call graphs constructed from bytecode have on average 1.7x as many reachable methods and 4.4x as many call edges as the call graphs constructed by analyzing Scala source. In other words, analyzing generated bytecode incurs a very large loss in precision.

Investigating further, we found that the most significant cause of precision loss is due to apply methods, which are generated from closures. These account for, on average, 25% of the spurious call edges computed by RTA^{wala} but not by TCA^{bounds}. The second-most significant cause of precision loss are toString methods, which account for, on average, 13% of the spurious call edges.

The ENSIME program is an interesting special case because it uses Scala constructs that are translated into code that uses reflection (see Section 3.5). As a result, the RTA^{wala} analysis makes conservative approximations that cause the call graph to become extremely large and imprecise[11]. This further reaffirms that a bytecode-based approach to call graph construction is highly problematic.

RQ2. To answer this question, we compare TCA^{names} and TCA^{bounds} and find that name-based analysis incurs a very significant precision loss: The call graphs generated by TCA^{names} have, on average, 10.9x as many call edges as those generated by TCA^{bounds}. Investigating further, we found that, on average, 66% of the spurious call edges computed by the name-based analysis were to apply methods, which implement closures.

RQ3. To answer this question, we compare TCA^{bounds} and TCA^{expand}. On the smaller benchmark programs that make little use of abstract types, the two produce identical results. Since KIAMA, PHANTM, and SEE contain some call sites on receivers with abstract types, TCA^{expand} computes more precise call graphs for them. For SCALAXB, SCALISP, and SQUERYL, call graph precision is not improved despite the presence of abstract types because the call sites on abstract receivers occur in unreachable code.

RQ4. To answer the fourth research question, we compare the TCA^{expand} and TCA$^{expand\text{-}this}$ analyses. In general, we found that the precision benefit of the special handling of this calls is small and limited to specific programs. In particular, we found that the number of call edges is reduced by 5% on KIAMA and by 1% on SEE, but that there is no significant difference on the other benchmarks. The situation for KIAMA is interesting in that TCA^{expand} finds 3.7% more instantiated types than TCA$^{expand\text{-}this}$. Those types are instantiated in methods found unreachable by TCA$^{expand\text{-}this}$.

The two most common reasons why the more precise rule TCA$^{expand\text{-}this}_{\text{THIS-CALL}}$ may fail to rule out a given call graph edge are that the caller M actually is inherited

[11] The summarized call graph computed by RTA^{wala} shown in Table 2 has 4,525 nodes and 61,803 edges. However, the size of the call graph originally computed by RTA^{wala} (before summarizing the library code) has 78,901 nodes and 7,835,170 edges. We experimentally confirmed that nearly half of these edges are in parts of the libraries related to the reflection API.

Table 3. The time (in seconds) taken by RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, TCA$^{expand\text{-}this}$, and RTA^{wala} to compute the call graphs.

	RA	**TCA**^{names}	**TCA**^{bounds}	**TCA**^{expand}	**TCA**$^{expand\text{-}this}$	**RTA**^{wala}	**scalac**
ARGOT	4	3.4	3.2	3.5	3.5	11.3	25.3
ENSIME	32.1	24.8	25	29	27.5	510.2	60.6
FIMPP	5.5	4.9	7.4	7.5	8	14.3	36.1
KIAMA	286	83	125.6	132.9	115.3	66.9	104.1
PHANTM	55.4	43.2	51.1	54.3	52.5	26.8	70.2
SCALAXB	113.4	16.3	10.9	11.5	12.7	21.1	85.9
SCALISP	3	2.9	3	3.1	3.2	12.6	25.6
SEE	6.9	6.3	8.2	8.1	8.8	13.9	40
SQUERYL	21	11.5	5.6	6.3	6.8	20.9	61.6
TICTACTOE	1.7	1.7	1.9	2	2	9.9	16.3

into the run-time receiver type `C`, so the call can occur, or that the caller `M` can be called through `super`, so using the rule would be unsound, as explained in Section 5.1. Across all the benchmark programs, the rule failed to eliminate a call edge at 80% of call sites on `this` due to the caller `M` being inherited into `C`, and at 15% of call sites on `this` due to the caller `M` being called through `super`.

RQ5. The running times of the analyses are presented in Table 3. For comparison, the last column of the table also shows the time required to compile each benchmark using the unmodified Scala compiler. Although our implementation has not been heavily tuned for performance, the analysis times are reasonable compared to `scalac` compilation times. The high imprecision of the RA analysis generally makes it significantly slower than the other, more complicated but more precise analyses. The TCA^{names} analysis is sometimes significantly faster and sometimes significantly slower than the TCA^{bounds} analysis, since it avoids the many expensive subtype tests, but is significantly less precise. The TCA^{expand} and TCA$^{expand\text{-}this}$ analyses have generally similar execution times as the TCA^{bounds} analysis because abstract types and `this` calls are a relatively small fraction of all call sites in the benchmark programs.

The long running time of nearly 500 seconds of RTA^{wala} on ENSIME is because the computed call graph becomes extremely large (see discussion of RQ1).

RQ6. Certain applications of call graphs require call sites to have a unique outgoing edge. For example, whole-program optimization tools [28] may inline such "monomorphic" call sites. It is therefore interesting to measure the ability of the different algorithms to resolve call sites to a unique target method. Table 4 shows, for each benchmark program, the number of monomorphic and

Table 4. Number of monomorphic and polymorphic reachable call sites in the summarized version of call graphs computed using RA, and how many of them became unreachable, monomorphic, or polymorphic in TCA$^{expand\text{-}this}$.

			TCA$^{expand\text{-}this}$		
		RA	**Unreachable**	**Mono**	**Poly**
ARGOT	**Mono**	1,200	459	741	-
	Poly	1,296	575	687	34
ENSIME	**Mono**	10,901	398	10,503	-
	Poly	8,433	430	7,545	458
FIMPP	**Mono**	4,058	56	4,002	-
	Poly	1,636	7	1,478	151
KIAMA	**Mono**	40,974	15,103	25,871	-
	Poly	27,869	11,586	15,337	946
PHANTM	**Mono**	17,500	1,023	16,477	-
	Poly	18,611	631	16,387	1,593
SCALAXB	**Mono**	22,170	12,206	9,964	-
	Poly	24,809	17,181	7,083	545
SCALISP	**Mono**	1,163	143	1,020	-
	Poly	1,106	154	890	62
SEE	**Mono**	5,327	258	5,069	-
	Poly	4,126	321	2,998	807
SQUERYL	**Mono**	6,453	4,092	2,361	-
	Poly	6,369	4,794	1,498	77
TICTACTOE	**Mono**	330	1	329	-
	Poly	204	0	187	17

polymorphic call sites, as determined by the RA analysis. The table also shows how these calls are resolved by the TCA$^{expand\text{-}this}$ analysis. For example, for ENSIME, the RA analysis finds 10,901 monomorphic calls and 8,433 polymorphic calls. Of the 10,901 calls that are identified as monomorphic by RA, 398 are identified as unreachable by the more precise TCA$^{expand\text{-}this}$ analysis and the remaining 10,503 remain as monomorphic calls. More interestingly, of the 8,433 calls that RA identifies as polymorphic, 430 become unreachable, 7,545 become monomorphic, and only 458 remain polymorphic according to TCA$^{expand\text{-}this}$.

7 Conclusions

We presented a family of low-cost algorithms for constructing call graphs of Scala programs, in the spirit of Name-Based Resolution (RA) [26], Class Hierarchy

Analysis (CHA) [9] and Rapid Type Analysis (RTA) [6]. Our algorithms consider how traits are combined in a Scala program to improve precision and handle the full Scala language, including features such as abstract type members, closures, and path-dependent types. Furthermore, we proposed a mechanism for resolving calls on the this reference more precisely, by considering overriding definitions of the method containing the call site.

We implemented the algorithms in the context of the Scala compiler, and compared their precision and cost on a collection of Scala programs. We found that TCA^{names} is significantly more precise than RA, indicating that maintaining a set of instantiated trait combinations greatly improves precision. Furthermore, TCA^{bounds} is significantly more precise than TCA^{names}, indicating that subtyping-based call resolution is superior to name-based call resolution. The improvements of TCA^{expand} over TCA^{bounds} occur on a few larger subjects that make nontrivial use of abstract type members and type parameters. Similarly, TCA$^{expand\text{-}this}$ only did significantly better than TCA^{expand} on programs that make nontrivial use of subtyping and method overriding.

Prior to our work, if one needed a call graph for a Scala program, the only available method was to analyze the JVM bytecodes produced by the Scala compiler. Since significant type information is lost during the compilation process, RTA call graphs constructed from the JVM bytecodes can be expected to be much less precise than the call graphs constructed using our new algorithms, as is confirmed by our experimental results.

While our research has focused on Scala, several aspects of the work are broadly applicable to other statically typed object-oriented languages. In particular, the special handling of calls on this can be integrated with existing algorithms such as CHA and RTA for languages such as Java, C#, and C++.

Acknowledgments. We are grateful to Max Schäfer and the anonymous ECOOP reviewers for many invaluable comments and suggestions, and to Rob Schluntz for assistance with testing. This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Ontario Ministry of Research and Innovation.

References

1. Agesen, O. Constraint-based Type Inference and Parametric Polymorphism. In *SAS* (1994), pp. 78–100.
2. Ali, K., and Lhoták, O. Application-only Call Graph Construction. In *ECOOP* (2012), pp. 688–712.
3. Ali, K., and Lhoták, O. Averroes: Whole-program analysis without the whole program. In *ECOOP* (2013), pp. 378–400.
4. Ali, K., Rapoport, M., Lhoták, O., Dolby, J., and Tip, F. Constructing call graphs of Scala programs. Tech. Rep. CS-2014-09, U. of Waterloo, 2014.
5. Bacon, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California, Berkeley, 1997.
6. Bacon, D. F., and Sweeney, P. F. Fast static analysis of C++ virtual function calls. In *OOPSLA* (1996), pp. 324–341.

7. Bravenboer, M., and Smaragdakis, Y. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA* (2009), pp. 243–262.
8. Cremet, V., Garillot, F., Lenglet, S., and Odersky, M. A core calculus for Scala type checking. In *MFCS* (2006), pp. 1–23.
9. Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP* (1995), pp. 77–101.
10. DeFouw, G., Grove, D., and Chambers, C. Fast Interprocedural Class Analysis. In *POPL* (1998), pp. 222–236.
11. Grove, D., and Chambers, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (2001), 685–746.
12. Heintze, N. Set-Based Analysis of ML Programs. In *LISP and Functional Programming* (1994), pp. 306–317.
13. Heintze, N., and Tardieu, O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI* (2001), pp. 254–263.
14. Henglein, F. Dynamic Typing. In *ESOP* (1992), pp. 233–253.
15. IBM. T.J. Watson Libraries for Analysis WALA. `http://wala.sourceforge.net/`, April 2013.
16. Kneuss, E., Suter, P., and Kuncak, V. Phantm: PHP analyzer for type mismatch. In *SIGSOFT FSE* (2010), pp. 373–374.
17. Lhoták, O., and Hendren, L. J. Scaling Java Points-to Analysis Using SPARK. In *CC* (2003), pp. 153–169.
18. Lhoták, O., and Hendren, L. J. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC* (2006), pp. 47–64.
19. Odersky, M. The Scala Language Specification version 2.9. Tech. rep., EPFL, May 2011. DRAFT.
20. Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*, 2nd ed. Artima Press, 2012.
21. Ryder, B. Constructing the call graph of a program. *IEEE Transactions on Software Engineering 5*, 3 (1979), 216–226.
22. Sallenave, O., and Ducourneau, R. Lightweight generics in embedded systems through static analysis. In *LCTES* (2012), pp. 11–20.
23. Sewe, A., Mezini, M., Sarimbekov, A., and Binder, W. Da capo con scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *OOPSLA* (2011), pp. 657–676.
24. Shivers, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991.
25. Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP* (2012), pp. 435–458.
26. Srivastava, A. Unreachable procedures in object oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.
27. Tip, F., and Palsberg, J. In *OOPSLA* (2000), pp. 281–293.
28. Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst. 24*, 6 (2002), 625–666.
29. Vallée-Rai, R., Gagnon, E., Hendren, L. J., Lam, P., Pominville, P., and Sundaresan, V. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *CC* (2000), pp. 18–34.

A Artifact Description

Authors of the artifact. Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip

Summary. The artifact is based on the implementation of the Scala source-level call graph construction algorithms we have discussed in this paper, RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, and TCA$^{expand\text{-}this}$. Additionally, the artifact includes the implementation of the bytecode-based call graph construction algorithm, RTA^{wala}. The source-level algorithms are implemented as a Scala compiler plugin. On the other hand, RTA^{wala} is the state-of-the-art implementation of the RTA algorithm provided by the WALA framework. The provided artifact package is designed to support the repeatability of the experiments of the paper. In particular, it allows users to generate the call graphs for a variety of benchmarks using one or more of the six algorithms. Instructions for the general use of our Scala compiler call graph plugin "scalacg" with any Scala source code are also provided.

Content. The artifact package includes:

- `scalabench.tar.gz`: all the necessary scripts, runnable JARs, required to replicate our experiments.
- `scalacg.tar.gz`: the source code of our Scala compiler plugin.
- `callgraph-plugin.jar`: our Scala compiler plugin. Additionally, it contains the ProBe tool to compare and visualize call graphs.
- `walacg.jar`: a runnable JAR for the implementation of RTA^{wala}.
- `index.html`: detailed instructions for using the artifact.

We provide a VirtualBox appliance containing Ubuntu 13.10 (Saucy Salamander), fully configured to simplify repeatability of our experiments. The image includes the file `scalabench.tar.gz` on the desktop.

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of our code and runnable JAR files are available at: `http://plg.uwaterloo.ca/~karim/projects/scalacg`.

Tested platforms. The artifact is known to work on any platform running Oracle VirtualBox version 4 (`https://www.virtualbox.org/`) with at least 8 GB of free space on disk and at least 16 GB of free space in RAM. However, the Scala compiler call graph plugin itself is known to work on any platform running Scala 2.10.2, though it has been only tested on Linux and Mac OS X environments.

License. EPL-1.0 (`http://www.eclipse.org/legal/epl-v10.html`), except for any external packages, tools, sources (including benchmark sources) used in the artifact, those respect their original licenses.

MD5 sum of the artifact. b92d617c9636a0022eac665595982386

Size of the artifact. 5.3 GB